**Logic before coding (work out problem write pseudocode)**

**Naming convention (See handout)**

**Small tasks (Avoid large functions, each method should achieve a small task)**

**Comment your code (comments relating to methods e.g. Javadoc style)**

**Indentation (You learned this in first year)**

**Testing all functionality (Unit testing…….driver programs)**

**Code**

* Use plenty of white space to clarify code.
* Indent properly to show structure, and be consistent with your indentation throughout the entire program.
* Put statements ending with semicolons and { } on separate lines (except for loops).

**for (i=o;i<5;i++)**

**{ cout<<i\*j; j=j-3; }**

**for (i = 0; i < 5; i++){**

**cout << i \* j;**

**j = j - 3;**

**} //end for**

**Variables / Constants**

* Chose variable names and constants that are self-documenting. Use only single letter variables for counting.

*Example:* **c = a - b;** vs. **score = total\_shots - total\_missed;**

* Constants should generally be all capitalized, and variables should generally be all lower case (camel-casing is also acceptable for variables; just be consistent).

*Example:* **const int NAME\_LEN = 4;** and **int num\_of\_students = 20**; **// or numOfStudents**

* Use constants for all “magic numbers” or whenever a particular value is used more than once.

**Documentation**

* Always place your name, date, and program description at the top of the program.
* Write comments that express more of the *why* than the *how*. Document the code’s intent.

*Example:* **// Divide total grades by total students**

vs. **// Compute student grade average**

* Make your comments say something about the code that the code can’t say about itself.

*Example:* **level++; // Add 1 to level**

vs. **level++; // Advance to next level before processing remaining lives**

* Typically a comment should refer to a logical grouping of lines rather than a single line of code.
* Avoid all but the most common abbreviations in comments so they’re easy to read.
* Clearly separate comments from code.
* Use a commenting style that is not overly tedious, time-consuming, or a maintenance monster.
* Outline your code with comments before you write it rather than doing all of the documentation at the end. You’ll save yourself time from having to figure out tricky places or forgetting details, assumptions, and subtleties of design.

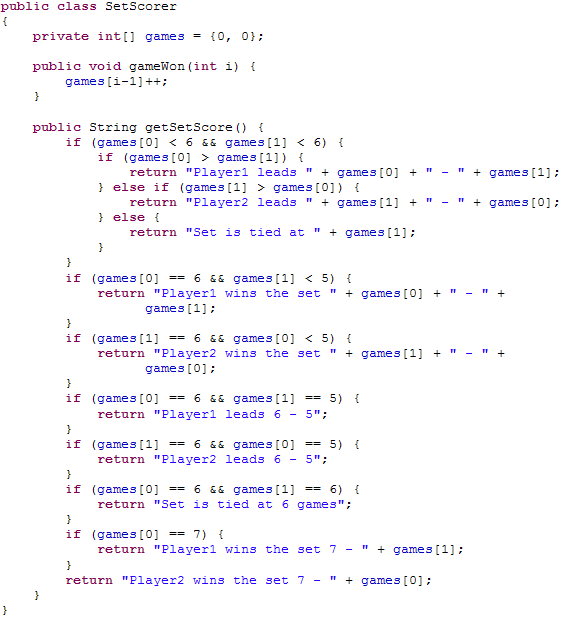
**Methods**

* method names should usually consist of a verb and a noun which describe the function’s purpose. *Example:* **cms()** vs. **calcManagerSalary()**
* A method should have a single purpose. *Example:* Figuring an average, printing a student list, and getting input in one method vs. three functions which perform each of those actions.
* Use method to reduce redundancy in your programs. If you have three code segments that do nearly the exact the same thing, that code is a good candidate for a method.

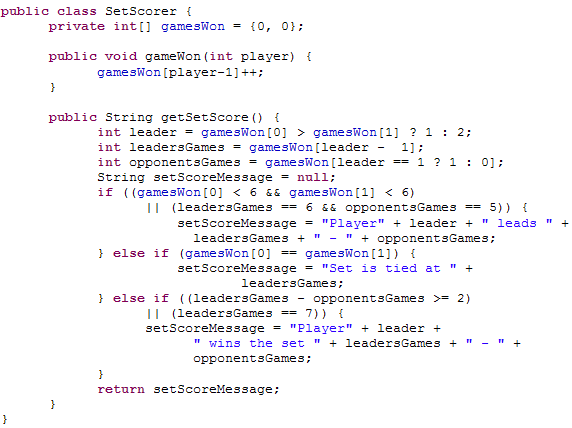
**Elegant Code**

There are many things that contribute to clean code. Some are universal and apply to any type of programming language or problem domain for which you are developing software. Some of the properties of clean code depend upon the specific programming language.

Let's take a quick look at two short pieces of code that do the same thing. This is code that scores sets in tennis. Each one has a method to record a game won by a player and another method to return a message containing the score. Both assume that all inputs are valid and no checks for invalid state are made. They are written in a design-by-contract method, where the burden on the correct use of the class is on the client. They both pass the same set of unit tests. Now, take a look at the two listings below. Which do you think is "cleaner," Listing 1 or Listing 2?



**Listing 1: First tennis set scoring example**



**Listing 2: Second tennis set scoring example**

Listing 1 is written in a naive style that a beginning programmer might use with code that seems to repeat itself. It's not necessarily complex, but seems cluttered to me and quite a bit inefficient. Listing 2 has more complex conditions, but if you understand Java, it reads quite well. The only part that you might question is the first part of the condition in the last *else if*. It turns out that when you get to this clause, one player has won.

Neither implementation is wrong. In fact, they are both small enough that they're not much more than a example, so talking about how clean the code isn't a very useful discussion to have about these listings, except to whet your appetite about what makes one implementation cleaner than another.

Source : http://www.ibm.com/developerworks/rational/library/nov06/pollice/